

---

# Higher Order Functions

Mattox Beckman

beckman@iit.edu

Illinois Institute of Technology

## Objectives

---

## §0 Objectives

The purpose of this lecture is to give you an introduction to higher order functions. As a result of this lecture, you should...

- know how to create higher order functions
- understand the concept of *closures*
- know how to use and write the following kinds of higher order functions:
  - function combination — `twice`, `compose`
  - interface changes — `curry`, `uncurry`
  - list processing — `fold_right`, `map`, `zip_with`

A type is said to be *first class* type when it can be

- **assigned** to a variable, **passed** as a parameter, or **returned** as a result

Examples:

- APL: scalars, vectors, arrays
- C: scalars, pointers, structures
- C++: like C, but with classes
- Scheme, Lisp, ML: scalars, lists, tuples, functions

**The Kind of Data a Program Manipulates Changes the Expressive Ability of a Program**

# Compose

## §1 First Class Values

---

```
1 # let double x = x * 2;;
2 val double : int -> int = <fun>
3 # let inc x = x + 1;;
4 val inc : int -> int = <fun>
5 # let compose f g = fun x -> f (g x);;
6 val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b
```

- These are *higher order types*
- Important skill: be able to tell from the *type* of the function what action it performs.
- Try combining `compose`, `inc`, and `double`. What happens?

**Twice****§1 First Class Values**

---

```
1 # let twice f x = f ( f x );;  
2 val twice : ('a -> 'a) -> 'a -> 'a = <fun>  
3 # twice inc 2;;  
4 - : int = 4
```

Can you write `twice` using `compose`?

## Sectioning

---

### §2 Closures

- A function that can create another function is a HOF.
- The simplest way to make one is to pass in only some of the arguments a function requires.

How can you interpret the type `int -> float -> string`?

- When this happens to operators, it's called *sectioning*.

```
1 # (+) ; ;  
2 - : int -> int -> int = <fun>  
3 # let inc = (+) 1 ; ;  
4 val inc : int -> int = <fun>
```

**Free Variables****§2 Closures**

---

- Consider what happens here....

```
1 # let plus x y = x + y;;  
2 val plus : int -> int -> int = <fun>  
3 # let inc = plus 1;;  (* no parameter for y *)  
4 val inc : int -> int = <fun>  
5 # inc 5;;  
6 - : int = 6  
7 # let x = 20;;  
8 val x : int = 20
```

What will be the result of `inc 5` now?

## Closures

## §2 Closures

- A *closure* is an expression (usually a function) along with an environment.

```
1 # let plus x y = x + y;;  
2 val plus : int -> int -> int = <fun>  
3 # let inc = plus 1;; (* no parameter for y *)  
4 val inc : int -> int = <fun>
```

$$\text{inc} = \triangleleft \{x \mapsto 1\}, \text{fun } y \Rightarrow x + y \triangleright$$

- The “local” environment will have precedence over any other variables that exist.
- Note that no computation is done at all until all arguments are received.



## Two Isomorphic Types

---

### §3 Interfaces

● What is the difference between these two types?

● `int -> int -> int`

● `(int * int) -> int`

```
1 # let foo x y = x + y;;  
2 val foo : int -> int -> int = <fun>  
3 # let bar (x,y) = x + y;;  
4 val bar : int * int -> int = <fun>
```

● What will happen if we try to section these functions?

## Curried Functions

---

## §3 Interfaces

- Type `int -> int -> int` is said to be in *curried* form.
- What will the following function do? You are only given the type.

```
1 # let foo secret stuff  
2 val foo : ('a * 'b -> 'c) -> 'a -> 'b -> 'c = <fun>
```

- Can you figure out how to write this function?

- Here are the function definitions. You should memorize them.

```
1 # let curry f a b = f (a,b);;
2 val curry : ('a * 'b -> 'c) -> 'a -> 'b -> 'c = <fun>
3 # let uncurry f (a,b) = f a b;;
4 val uncurry : ('a -> 'b -> 'c) -> 'a * 'b -> 'c = <fun>
```

- Now we can section an uncurried function....

```
1 # let bar (x,y) = x + y;;
2 val bar : int * int -> int = <fun>
3 # let inc = (curry bar) 1;;
4 val inc : int -> int = <fun>
```

- We can also do other weird things....

```
1 # let flip f a b = f b a;;
2 val flip : ('a -> 'b -> 'c) -> 'b -> 'a -> 'c = <fun>
3 # let (-) = flip (-);;
4 val ( - ) : int -> int -> int = <fun>
5 # 2 - 5;;
6 - : int = 3
```

## Combining Lists and Functions

### §4 Lists and Functions

---

- You can make *lists of functions*.
- Note that we don't have to know in advance how many functions we will be processing.
- Why isn't this type as general as `compose`?  
 $( 'a \rightarrow 'b ) \rightarrow ( 'c \rightarrow 'a ) \rightarrow 'c \rightarrow 'b$

```
1 # let rec complist flst x =  
2   match flst with  
3   | [] -> x  
4   | f::fs -> f (complist fs x);;  
5 val complist : ('a -> 'a) list -> 'a -> 'a = <fun>  
6 # complist [inc; double; inc] 4;;  
7 - : int = 11
```

**Repeating a function  $n$  times****§4 Lists and Functions**

---

- We do not have to know in advance how many times we will process them.

```
1 # let rec fnth n f x =  
2   match n with  
3   | 0 -> x  
4   | _ -> f (fnth (n-1) f x);;  
5 val fnth : int -> ('a -> 'a) -> 'a -> 'a = <fun>  
6 # fnth 5 inc 2;;  
7 - : int = 7  
8 # fnth 10 double 10;;  
9 - : int = 10240
```

**Mapping functions the hard way**

Consider the following definitions. What do they have in common?

```
1 # let rec inclist lst = match lst with
2   | [] -> []
3   | x::xs -> x + 1 :: inclist xs;;
4 val inclist : int list -> int list = <fun>
5 # let rec doublelist lst = match lst with
6   | [] -> []
7   | x::xs -> x * 2 :: doublelist xs;;
8 val doublelist : int list -> int list = <fun>
9 # inclist [2;3;4];;
10 - : int list = [3; 4; 5]
11 # doublelist [2;3;4];;
12 - : int list = [4; 6; 8]
```

## **Mattox's Rule of Computing**

---

**The computer exists to work for us; not us for the computer. If you are doing something repetitive for the computer, you are doing something wrong.**

**Stop what you're doing and find out how to do it right.**



## Mapping functions the easy way

$$\text{map } f \ [x_1; x_2; \dots x_n] = [f(x_1); f(x_2); \dots f(x_n)]$$

```
1 # let rec map f lst = match lst with
2   | [] -> []
3   | x::xs -> f x :: map f xs;;
4 val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
5 # map inc [2;3;4];;
6 - : int list = [3; 4; 5]
7 # map double [2;3;4];;
8 - : int list = [4; 6; 8]
```

- `inc` and `double` have been transformed into recursive functions.
- How would you have done this in C++ or Java?

**Related function: zip**

```
1 # let rec zip f alst blst =  
2   match alst,blst with  
3   | [],_ -> []  
4   | _,[] -> []  
5   | a::aa, b::bb -> f a b :: zip f aa bb;;  
6 val zip : ('a -> 'b -> 'c) -> 'a list -> 'b list  
7           -> 'c list = <fun>  
8 # zip (+) [2;3;4] [5;6;8];;  
9 - : int list = [7; 9; 12]  
10 # zip (fun a b -> a * a + b * b) [2;3;4] [3;4;5];;  
11 - : int list = [13; 25; 41]
```

🔴 Side note: `zip` is sometimes known as `map2` or `zipwith`.

**Folding functions****§6 Folding functions**

---

Consider the following definitions. What do they have in common?

```
1 # let rec sumlist lst = match lst with
2   | [] -> 0
3   | x::xs -> x + sumlist xs;;
4 val sumlist : int list -> int = <fun>
5 # let rec prodlist lst = match lst with
6   | [] -> 1
7   | x::xs -> x * prodlist xs;;
8 val prodlist : int list -> int = <fun>
9 # sumlist [2;3;4];;
10 - : int = 9
11 # prodlist [2;3;4];;
12 - : int = 24
```

**fold****§6 Folding functions**

---

$$\text{fold } f \ [x_1; x_2; \dots x_n] \ z = f(x_1, f(x_2, f(\dots, z)))$$

```
1 # let rec fold_right f lst z = match lst with
2   | [] -> z
3   | x::xs -> f x (fold_right f xs z);;
4 val fold_right : ('a -> 'b -> 'b) -> 'a list
5                 -> 'b -> 'b = <fun>
6 # fold_right (+) [2;3;4] 0;;
7 - : int = 9
8 # fold_right (fun a b -> a * a + b) [2;3;4] 0;;
9 - : int = 29
```

- To use `fold`, we specify the function *and* the base case.

# Encoding Recursion using fold

## §6 Folding functions

```
1 # let rec flatten lst = match lst with
2   | [] -> []
3   | x::xs -> x @ flatten xs;;
4 val flatten : 'a list list -> 'a list = <fun>
5 # flatten [ [2;3]; [3;4;5]; [6;0] ];;
6 - : int list = [2; 3; 3; 4; 5; 6; 0]
```

Operation

Recursive Call

Base Case

```
1 let flatten lst =
2   fold_right (fun x y -> x @ y) lst [];;
```

- **Note well:** the second parameter of the function argument to `fold_right` represents the *result* of the recursive call.

**Another example****§6 Folding functions**

---

- You can write `map` using `fold`.

```
1 # let fmap f lst =  
2     fold_right (fun x y -> f x :: y) lst [];  
3 val fmap : ('a -> 'b) -> 'a list -> 'b list = <fun>  
4 # fmap inc [2;3;4];;  
5 - : int list = [3; 4; 5]
```

- Note, the reverse direction will not work. Why not?

**Problems****§7 Activity**

---

1. Write a function `flipuc` that flips uncurried functions. Do this using only `flip`, `curry`, and `uncurry`.
2. Write the function that has the type  
 $( 'a \rightarrow 'b ) \rightarrow 'a * 'c \rightarrow 'b$
3. Use `fold_right` to write a function that takes a list and then returns it.
4. Use `fold_right` to write a function that takes a list and removes all elements less than zero.

**Problem 1****§7 Activity**

---

- Write a function `flipuc` that flips uncurried functions. Do this using only `flip`, `curry`, and `uncurry`.

```
1 # let flipuc f = uncurry (flip (curry f));;
2 val flipuc : ('a * 'b -> 'c) -> 'b * 'a -> 'c = <fun>
3 # let sub (a,b) = a - b;;
4 val sub : int * int -> int = <fun>
5 # let sub = flipuc sub;;
6 val sub : int * int -> int = <fun>
7 # sub (5, 2);;
8 - : int = -3
```



**Problem 2****§7 Activity**

---

- Write the function that has the type  $( 'a \rightarrow 'b ) \rightarrow 'a * 'c \rightarrow 'b$

```
1 # let prol f (x,y) = f x;;
2 val prol : ('a -> 'b) -> 'a * 'c -> 'b = <fun>
3 # prol inc (4,8);;
4 - : int = 5
5 # prol inc (4,"hi");;
6 - : int = 5
```

**Problem 3****§7 Activity**

---

- Use `fold` to write a function that takes a list and then returns it.

```
1 # let cons a b  = a :: b;;  
2 val cons : 'a -> 'a list -> 'a list = <fun>  
3 # let return lst = fold_right cons lst [];;  
4 val return : 'a list -> 'a list = <fun>  
5 # return [2;3;4];;  
6 - : int list = [2; 3; 4]
```

Yeah, it's pretty boring, but knowing the values that make the identity is important.

**Problem 4****§7 Activity**

---

- Use `fold` to write a function that takes a list and removes all elements less than zero.

```
1 # let gtzero lst =  
2     fold_right (fun a b -> if a >= 0  
3                           then a :: b  
4                           else b) lst [];;  
5 val gtzero : int list -> int list = <fun>  
6 # gtzero [2;-3;4;-65];;  
7 - : int list = [2; 4]
```

A related function `filter` does the same thing, but takes a parameter for the `a >= 0` part.